SciSpot: A Framework for Low-cost Scientific Computing On Transient Cloud Servers

Paper # 105

ABSTRACT

In this paper, we...

1 INTRODUCTION

Scientific computing applications are a crucial component in the advancement of science and engineering, and play an important role in the analysis and processing of data, and understanding and modeling natural processes. These applications are typically implemented as large-scale parallel programs that use parallel-computing communication and coordination frameworks such as MPI. To take advantage of their parallel nature, conventionally, these applications have mostly been deployed on large, dedicated high performance computing infrastructure such as super computers.

(Vikram: Application of computer simulation plays a critical role in understanding natural and synthetic phenomena associated with a wide range of material, biological, and engineering systems. This scientific computing approach involves the analysis and processing of data generated by the mathematical model representations of these systems implemented on computers. Typically, these scientific computing applications (simulations) are designed as parallel programs that leverage the communication and coordination frameworks associated with parallel computing techniques such as MPI in order to yield useful information at a faster pace (shorter user time). To exploit the parallel processing capabilities, such applications are routinely deployed on large, dedicated high performance computing infrastructure such as supercomputers [? ? ?].)

Increasingly, cloud computing platforms have begun to supplement *cite* and complement *cite; how true is this?* conventional HPC infrastructure ~~in order~~ to meet the large computing and storage requirements of scientific applications. Public cloud platforms such as Amazon's EC2, Google Cloud Platform, and Microsoft Azure, offer multiple benefits such as *on-demand* resource allocation, convenient pay-as-you-go pricing models, ease of provisioning and deployment, and near-instantaenous elastic scaling. Most cloud platforms offer *Infrastructure as a Service*, and provide computing resources in the form of *Virtual Machines (VMs)*, on which a wide range of applications such as web-services, distributed data processing, distributed machine learning, etc., are deployed.

(Vikram: The extensive use of cloud platforms to host and run a wide range of applications such as web-serives and distributed data processing have inspired early investigations in the direction of using these resources for scientific computing applications to meet the large computing and storage requirements of the latter. Early work in this area has shown the potential of using these cloud platforms to both supplement and complement conventional HPC infrastructure. Public cloud platforms such as Amazon's EC2, Google Cloud Platform, and Microsoft Azure, offer multiple benefits: *on-demand* resource allocation, convenient pay-as-you-go pricing models, ease of provisioning and deployment, and near-instantaenous elastic scaling, to name a few. Most cloud platforms offer *Infrastructure as a Service*, and provide computing resources in the form of Virtual Machines (VMs) that are application-agnostic and can serve as deployment sites for a wide range of applications such as web-services, distributed machine learning, and scientific simulations.)

, ,
.

~~In order to~~ To meet the diverse resource demands of different applications, public clouds offer resources (i.e., VM's) with multiple different resource configurations (~~such as~~ e.g., number of CPU cores~~, and~~ memory capacity), and pricing and availability contracts. Conventionally, cloud VMs have been offered with "on-demand" availability, such that the lifetime of the VM was solely determined by the owner of the VM (i.e., the cloud customer). Increasingly however, cloud providers have begun offering VMs with *transient*, rather than continuous on-demand availability. Transient VMs can be unilaterally revoked and preempted by the cloud provider, and applications running inside them face fail-stop failures. Due to their volatile nature, transient VMs are offered at steeply discounted rates. Amazon EC2 spot instances, Google Cloud Preemptible VMs, and Azure Batch VMs, are all examples of transient VMs, and are offered at discounts ranging from 50 to 90%.

However, deploying applications on cloud platforms presents multiple challenges due to the *fundamental* differences with conventional HPC clusters—which most applications still assume as their default execution environment. While the on-demand resource provisioning and pay-as-you-go pricing makes it easy to spin-up computing clusters in the cloud, for effective resource utilization, the deployment of applications must be cognizant of the heterogeneity in VM sizes, pricing, and availability. Crucially, optimizing for *cost*, and not just makespan, becomes an important objective in cloud deployments. Furthermore, although using transient resources can drastically reduce computing costs, their preemptible nature results in frequent job failures. Preemptions can be mitigated with additional fault-tolerance mechanisms and policies [9**?**], although they impose additional performance and deployment overheads.

In this paper, we develop principled approaches for deploying parallel scientific applications on the cloud at low cost, and present SciSpot, a system for deploying and orchestrating scientific applications on cloud transient servers. Our policies for tackling the resource heterogeneity and transient availability of cloud VMs build on a key insight: most scientific applications are deployed as a collection or "bag" of jobs. These bags of jobs represent multiple instantiations of the same computation with different parameters. For instance, each job may be running a (parallel) simulation with a set of simulation parameters, and different jobs in the collection run the simulation on a different set of parameters. Collectively, a bag of jobs can be used to "sweep" or search across a multi-dimensional parameter space to discover feasible and viable parameters.

Prior approaches and systems for mitigating transiency and cloud heterogeneity have largely targeted individual instantiations of jobs [9**?** –11]. For a bag of jobs, it is not necessary, or sufficient, to execute an individual job in timely manner—instead, we could selectively restart failed jobs in order to complete a *fraction* of jobs in a bag. Furthermore, treating the bag of jobs as a fundamental unit of computation allows us to select the "best" server configuration for a given application, by exploring different servers for initial jobs and running the reminder of the jobs on the optimal server configuration.

We show that optimizing across an entire bag of jobs and being cognizant of the relation between different jobs in a bag, can enable simple and powerful policies for optimizing cost, makespan, and ease of deployment. We implement these policies as part of the SciSpot framework, and make the following contributions:

(1) In order to select the "right" VM from the plethora of choices offered by cloud providers, we develop a search-based server selection policy that minimizes the cost of running applications. Our search based policy selects a transient server type based on it's cost, parallel speedup, and probability of preemption.
(2) Since transient server preemptions can disrupt the execution of jobs, we present the *first* empirical model and analysis of transient server availability that is *not* rooted in classical and out-dated bidding models for EC2 spot instances that have been proposed thus far. Our empirical model allows us to predict expected running and costs of jobs of different types and durations.
(3) We develop preemption-mitigation policies to minimize the overall makespan of bags of jobs, by taking into consideration the partial redundancy and relative "importance" of different jobs within a bag. Combined, our policies yield a cost saving of XXX% and a makespan reduction of XXX% compared to conventional cloud deployments, and a makespan reduction of XXX% compared to a conventional HPC supercomputer.
(4) Finally, ease of use and extensibility are one of the "first principles" in the design of SciSpot, and we present the design and implementation of the system components and present case studies of how scientific applications such as molecular dynamics simulations can be easily deployed on transient cloud VMs.

2 BACKGROUND

2.1 Transient Computing

2.2 Heterogeneity and Parallel Scientific Applications in the Cloud

2.3 Case Study: Molecular Dynamics Applications

Describe the kind of computation.

Scaling properties. Almost perfectly scalable with O(n) communication?

This can be a like a case study of parallel scientific simulations. Will help relate to parameters etc with more concrete examples.

Bag of jobs. Why multiple runs: parameter sweeps, search, or just multiple times to get confidence intervals and stable results in case of randomness.

3 UNDERSTANDING AND MODELING TRANSIENT SERVER PREEMPTIONS

Transient cloud servers, by their very nature have limited availability and are frequently preempted. These preemptions are akin to fail-stop failures, and are often preceeded by a small advance warning (few seconds) to allow for graceful shutdowns.

Since preemptions can impact the availability, performance, and cost of running applications, in this section, we examine their preemption characteristics. This modeling is important, because having a model of the availability can be useful in the context of predicting the running times of applications. Cloud providers offer a large number of servers of different configurations and types. Since transient server availability is fundamentally tied to supply and demand, the availability of servers of different types can be significantly different. Thus, selecting the "right" server type is crucial for minimizing the overall costs.

3.1 EC2 spot instances

The earliest form of transient cloud instances. In addition to having dynamic availability, also have dynamic pricing. "Classic" spot instances had price determining the availability, and thus a large amount of work was devoted to bidding and analyzing the prices.

However a recent change to the spot prices no longer allows these assumptions, rendering it impossible to obtain the *exact* availability information from the prices alone.

3.2 Google Preemptible VMs

Launched in 2015. Flat-rate discount of 80% compared to on-demand servers. Interesting availability SLA: the maximum lifetime is 24 hours, and can be preempted earlier as well.

In this paper we will look at these preemptible VMs and show how to model their availability. Given the inability to use EC2 prices, we believe that our approach is more generalizable and robust.

There are some distinguishing characteristics of GCP preemptible VMs that makes their failure modeling challenging. First is their flat pricing and no other signalling information about their preemption rates (MTBFs) that makes server selection difficult.

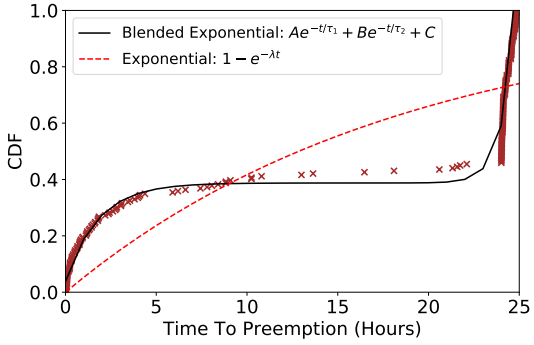

Figure 1: CDF of lifetimes of Google Preemptible Instances. Our blended exponential distribution fits much better than the conventional exponential failure distributions.

Modeling Failure Behavior of Preemptible VMs CDF is "sigmoid" shaped. $P = R * np.sinh((t - t0)/tau) + C$ with a very low $R = 10^{-6}, t_0 = 12, \tau = 0.9, C = 0.36$

Basically, this is a mixture of two distributions, the standard exponential distribution, which we call the stabilization rate and an exponentially increasing reclamation rate.

Preemptible VMs have three availability phases.

There are many early deaths, then a period of low failure rates, and then the failure rate is exponential with a positive exponent to enable the cloud provider to reclaim the VMs within the deadline (24 hours in the case of Google's Preemptible VMs).

3.3 Trade-offs in Server Selection

This presents us with many challenges in the cloud-deployment of these jobs.

Cloud providers offer multiple types of instances (VMs), with different hardware configuration (such as number of CPUs and memory size). The price of cloud servers is related to their hardware configuration, but it may not be strictly proportional to the hardware performance. For example, a VM with 32 CPUs may not be 32 times the cost of a single CPU VM.

For parallel and distributed applications, the type of servers selected has large implications on their performance. Consider the case of deploying an application on 8 8-core VMs vs. 16 4-core VMs. In both cases, the total number of CPU cores is the same. However, the larger number of VMs requires more communication between the application tasks, and thus may result in performance degradation. The performance of applications at different cluster configurations depends on their communication patterns and scaling properties.

Thus, when deploying applications on the cloud, one has to mindful of the cost and performance tradeoff. However, in the case of transient servers, the story does not stop here.

In addition to pricing differences, the transient availability of instances *also* differs by type. Because the availability of a transient VM is broadly determined by the overall supply and demand of the instances of that *particular* type, the "preemption rate" of VMs often depends on the type of the instance.

Thus, selecting a transient cloud server involves a complex tradeoff between the cost of servers, their performance, and the preemption-rate. We develop server selection policies in the next section.

4 SCISPOT DESIGN

Scientific simulation applications consume a large amount of computational resources, and are often used in the context of *exploratory* research, where a large amount of jobs are run with different simulation parameters. This can be either a *parameter sweep*, where a large number of parameters need to be evaluated, or a *search* over a large parameter space for the "right" set of parameters that yield the desired model behavior.

In this paper, we look at the problem of running scientific simulations on *transient* computing resources in public clouds.

Past work has largely been focused on running parallel jobs (such as MPI) in the cloud. However, considering entire *job-groups* or ensembles of jobs presents new challenges and opportunities in timely, low-cost computation.

Our system, SciSpot, is a unified framework for running large job-groups that result from parameter exploration.

Input and some assumptions: We assume that the job-group consists of $J_1 \ldots J_N$, with each job evaluating a model on some parameter. The list of parameters to explore can either be generated apriori (as in the case of parameter sweeps), or be dynamically generated as in the case of a search.

In this section, we will look at how we address these challenges:

(1) How to select the right type of cloud server for an application?
(2) How to effectively run job-groups?

SciSpot's key insight is considering an entire job-group can allow better and simpler optimizations that can be easily deployed.

Job-groups are executed in two phases. In the first phase, we search for the right type of server for the jobs in the job-group, and then in the second phase, we execute the remaining jobs on the chosen servers.

4.1 High-level flow

SciSpot is as a cost aware cloud resource manager for running large collections of parallel jobs. The collection forms a "bag" of jobs, which each job running the same executable but with different input parameters. As explained earlier, this is a common use case.

Running a large collection of computationally similar jobs permits many cost and performance optimizations. Given that cloud platforms offer a large number of resource configurations for their VMs, selecting the "right" VM for jobs can be especially beneficial in reducing the cost and running times. Running a large number of jobs with similar computation, communication, and runtime characteristics allows us to "explore" the right server configuration.

The execution of a bag of job kicks off with the user providing the executable, the expected resources requirements for a single job, and the fraction of jobs that must be completed.

Generating Jobs For a Bag: Optionally, SciSpot can also take as input a description of the (multi-dimensional) parameter space, and then generate the jobs. We do this with the users specifying the values that different job parameters can take, and then producing a list with all the permutations. Because job-failures can cause some parameter combinations to remain unexplored, we strive for uniformity of sampling by randomizing the order in which jobs in a bag are scheduled, so that we don't end up in a situation with a large fraction of any parameter remaining unexplored if the completion threshold has been met.

More details are provided in the Interface and Implementation section.

Therefore, SciSpot's execution of a bag of jobs is composed of two serial phases. In the first phase, we explore different cluster configurations to find the lowest cost server type. In the second phase, we run the reminder of the jobs in the bag on the servers of the selected type (the "exploitation" phase).

4.2 Exploration-based Server Selection

SciSpot's server selection policy seeks to identify the best server type for a given job-group.

As stated in the previous subsection, the transient server selection problem is challenging because it involves balancing multiple optimization criteria: applications want low cost, low preemptions, and high performance.

Server selection based on application characteristics is a subject of a growing amount of recent work. These approaches often use micro benchmarks to gather performance data of cloud servers, and then use application performance models to determine suitable VMs for a specific application. Another class of approaches uses "black box" performance modeling, where the application's performance is modeled

using a function of the resources, for example, by using linear regression.

In contrast to prior work, our server selection employs a "cold start" policy, and we do not run profiling or pilot jobs that can increase the overall running time and cost. Instead, we search for the "best" cluster configuration for jobs in a job-group, by exploring the cluster configuration space for the "optimum" server type that optimizes all the desired parameters: cost, running time, and revocation rate.

Thus, the first e jobs in the job group $J_1 \ldots J_e$ are the exploration jobs, run on different cluster configurations. We limit the total number of combinations to explore, by allowing users to submit an estimate of the total number of CPU cores that they desire for each job. This allows us to meet the user expectations in terms of performance and cost—whether the user expects us to spend a large amount of resources or not.

Thus, assuming that there are s different types of VM instances, the first s jobs are run on the s different types. Note that we use homogeneous clusters, since the performance of BSP programs in Heterogeneous environments can be degraded, and importantly, as we show, there are no performance or cost benefits to Heterogenity.

For each server type i, we calcuate the expected cost $E[C_i]$.

$E[C_i] = n_i * c_i * E[T_i]$, where c_i is the price (per second) of the server, n_i is the number of servers of that type required to meet the core-count requirement. The expected running time of the job depends on two factors: the actual running time T, and the increase in running time due to preemptions. Each preemption is akin to a fail-stop failure, which requires an application to restart. Our system makes no assumptions about the fault-tolerance policies supported by the application. For example, some applications may be able to *checkpoint* their state periodically. In either case (checkpointing or not), there is some work lost due to revocations. For ease of exposition, we assume no checkpointing. We discuss checkpointing in the next section (or never?)

Expected running time: Let the running time without failure be T:

$$E[T_i] = T + P(\text{at least one failure}) * T/2 \quad (1)$$

For calcuating the probability of failure, we assume that the failure rate of an individual server of the type is p_i.

$$P(\text{at least one failure}) = 1 - P(\text{no failure}) \quad (2)$$
$$= 1 - (1 - p_i)^{n_i} \quad (3)$$

Thus, we can see that if we select smaller VMs, we will require more of them (higher n_i), and this cluster configuration will have a larger probability of failure and thus higher running times and costs.

The probability of failure p_i depends on the type of server, and we use historically determined failure distributions. Roughly, if we assume exponentially distributed failures, then:

$$p_i = \frac{T}{\text{MTTF}_i} \quad (4)$$

Where MTTF is the mean time to failure of the server type, and T is the empirically determined job running time without failures (the best case).

In addition to searching over the servers, the effective number of servers is also dynamic in the case of transient environments due to preemptions. Thus, once we have found the appropriate server, we then explore the application's performance at smaller cluster sizes, which helps in the job-group policies that we discuss next.

4.3 Preemption-handling Policies

We run the remaining jobs on the right configuration that is determined through the server selection policy.

Our goal is to minimize costs given a deadline.

This determines two things: how many jobs should be run in parallel, and what to do upon a revocation.

The number of jobs in parallel determines the overall size of our cluster.

$$\text{number of parallel jobs} = \frac{N \cdot T}{\text{Deadline Duration}}$$

If a server is preempted, then the job running on it will cease to run. Our preemption handling policies then decide what to do:

(1) Restart the job on a smaller number of servers
(2) Replenish lost servers and restart job
(3) Discard job. This may be useful in case of parameter sweeps.

In addition, the user is also allowed to provide the fraction of jobs that are allowed to fail (η).

We exploit the naturally occuring intra-job redundancy.

New jobs. For new jobs, the questions are similar to the preemption policy ones, because they also must take into account the failures.

That is, assume that a job has finished and it is time to run a new job. So now, we have a set of servers that successfully ran a job. If we run on same set, then we may hit the 24 hour wall. But, if we discard these and launch new ones, then we face the infant mortality problem. So the question is, at what age should servers be retired? If they are too close to EOL (24 hours), then what's the point? Better start something fresh.

4.4 Checkpointing

Checkpointing requires the same number of servers, which may be tricky, whereas restarts can be on smaller number of nodes no problem.

Maybe talk about the dynamic programming based checkpointing here?

4.5 Early Stopping

Based on the energy function, we can stop some simulations early. The early stopping criteria helps in minimizing the number of jobs run to completion.

We use this to proactively monitor jobs, as well as to decide whether to restart a job if it is preempted.

This can be a fairly substantial section

5 CHECKPOINTING POLICIES

High rate of failure means that especially for long jobs, checkpointing is necessary. In this section, we develop checkpointing policies.

At a high level, we only checkpoint long jobs, based on the probability of preemption.

Once checkpointing for a job is enabled, we use proactive periodic checkpointing.

DMTCP is used for checkpointing MPI programs.

Conventionally, periodic checkpointing according to Young-Daly formula: $\tau = \sqrt{2 \cdot \delta \cdot \text{MTTF}}$. This assumes that failure arrival process is exponentially distributed.

However, this may not always hold true. Clearly, from Figure 1, the CDF is not exponential.

Much more strongly "bath-tub". Maybe can use exponentiated Weibull to model the failures.

Regardless, because it is not memoryless, the checkpointing interval cannot be uniform. Maybe can use [2], which uses dynamic programming and also comes with a simulator of sorts.

6 SCISPOT INTERFACE AND IMPLEMENTATION

Central controller.

Cloud APIs for launching the jobs.

Slurm.

Job groups are specified via a JSON file that we then use to generate different parameter combinations.

Example of a JSON file here? What about the description? Maybe some details about ranges and fixed values?

7 EVALUATION

The contenders:

(1) Run every job on un-tuned on-demand instance (cost and running time)
(2) Run every job on nanohub/big-red-2 (running and waiting time)
(3) Run on transient, restart every time (cost)
(4) SciSpot with early stopping and job sacrificing

Performance of 3 benchmarks on different types of instance types and bigred2.

7.1 Preemption likelihood curves

7.2 Searching for the best cloud configuration

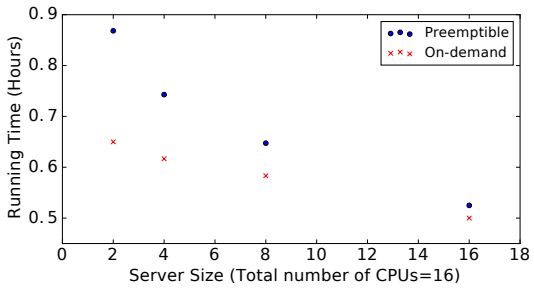

Figure 2: confinement running times

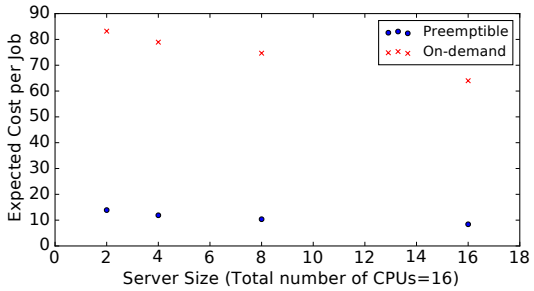

Figure 3: confinement cost

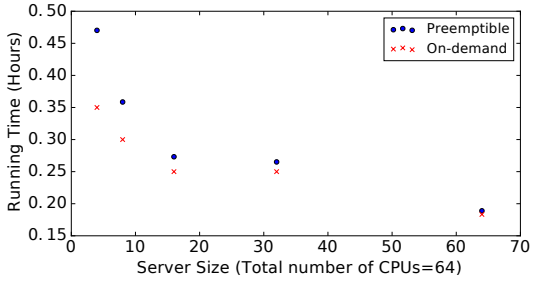

Figure 4: confinement running times

7.3 Total cost vs. running time graphs

7.4 HPC

8 RELATED WORK

8.1 Scientific applications on cloud

A classic survey is [8] [16]

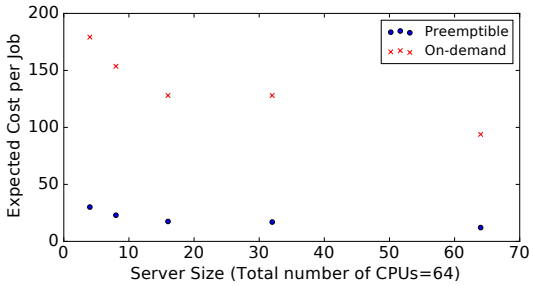

Figure 5: confinement cost

Figure 6: Ratio of waiting time to job running time on an HPC cluster. Average is 0.2

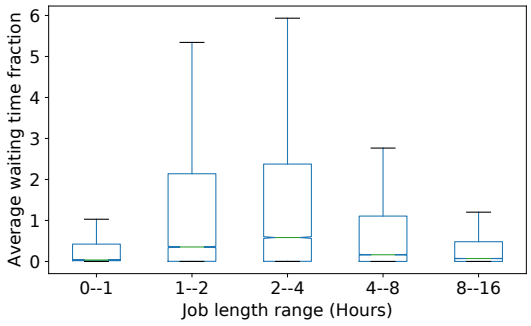

Figure 7: Waiting time fraction of jobs of different lengths varies.

Parameter sweep: [3]

Price optimizations for Scientific workflows in the cloud [5]

8.2 Transiency mitigation

[9] classic work on MPI and Spot. Uses checkpointing. Redundancy, but for what? User specified number of VMs. Does not do instance selection. BCLR for checkpointing.

MOre spot and MPI: [6]. FOcussed on bidding and checkpoint interval. But bidding doesnt matter.

[12] is early work for spot and MPI and

[11] a batch computing service

Heterogenity often used, but not useful in the context of MPI jobs [10]

Selecting the best instance type, often for data analysis computations [1], and [15], and others like Ernest and Hemingway.

All the past work was on EC2 spot market with gang failures and independent markets [6, 9]. However this assumption has now changed, and failures can happen anytime. Our failure model is more general, and applies to both cases.

8.2.1 Fault-tolerance for MPI. [4] has a discussion of checkpointing frequency which is comprehensive.

Replication is another way [13]

8.2.2 Huge amount of work on bidding in HPC. [14] [7]

8.3 Server Selection

Exploring a large configuration space using bayesian optimization methods in CherryPick [**?**] and Metis [**?**].

Can also use Latin Hypercube sampling for parameter exploration?

REFERENCES

[1] Alipourfard, O., and Yu, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. 15.

[2] Bougeret, M., Casanova, H., Rabie, M., Robert, Y., and Vivien, F. Checkpointing strategies for parallel jobs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11* (Seattle, Washington, 2011), ACM Press, p. 1.

[3] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)* (May 2000), pp. 349–363.

[4] Dongarra, J., Herault, T., and Robert, Y. Fault tolerance techniques for high-performance computing. 66.

[5] Garí, Y., Monge, D. A., Mateos, C., and García Garino, C. Learning budget assignment policies for autoscaling scientific workflows in the cloud. *Cluster Computing* (Feb. 2019).

[6] Gong, Y., He, B., and Zhou, A. C. Monetary cost optimizations for MPI-based HPC applications on Amazon clouds: checkpoints and replicated execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15* (Austin, Texas, 2015), ACM Press, pp. 1–12.

[7] Guo, W., Chen, K., Wu, Y., and Zheng, W. Bidding for Highly Available Services with Low Price in Spot Instance Market. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '15* (Portland, Oregon, USA, 2015), ACM Press, pp. 191–202.

[8] Iosup, A., Ostermann, S., Yigitbasi, M. N., Prodan, R., Fahringer, T., and Epema, D. H. J. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems 22*, 6 (June 2011), 931–945.

[9] Marathe, A., Harris, R., Lowenthal, D., De Supinski, B. R., Rountree, B., and Schulz, M. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *HPDC* (2014), ACM.

[10] Sharma, P., Irwin, D., and Shenoy, P. Portfolio-driven resource management for transient cloud servers. In *Proceedings of ACM Measurement and Analysis of Computer Systems* (June 2017), vol. 1, p. 23.

[11] Subramanya, S., Guo, T., Sharma, P., Irwin, D., and Shenoy, P. SpotOn: A Batch Computing Service for the Spot Market. In *SOCC* (August 2015).

[12] Taifi, M., Shi, J. Y., and Khreishah, A. SpotMPI: A Framework for Auction-Based HPC Computing Using Amazon Spot Instances. In *Algorithms and Architectures for Parallel Processing*, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds., vol. 7017. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 109–120.

[13] Walters, J. P., and Chaudhary, V. Replication-Based Fault Tolerance for MPI Applications. *IEEE Transactions on Parallel and Distributed Systems 20*, 7 (July 2009), 997–1010.

[14] Wolski, R., Brevik, J., Chard, R., and Chard, K. Probabilistic guarantees of execution duration for Amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17* (Denver, Colorado, 2017), ACM Press, pp. 1–11.

[15] Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing - SoCC '17* (Santa Clara, California, 2017), ACM Press, pp. 452–465.

[16] Zhai, Y., Liu, M., Zhai, J., Ma, X., and Chen, W. Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. In *State of the Practice Reports on - SC '11* (Seattle, Washington, 2011), ACM Press, p. 1.